INFO0947: Prefixe and Suffixe

Groupe 33: Aleksandr Pavlov,

Table des matières

1 Introduction

Dans le cadre du cours INFO-0947 nous avons du résoudre un problème donné et en créer un algorithm en C capable de : trouver la longueur du plus grand sous-tableau qui soit a la fois le préfixe et le suffixe d'un tableau donné. Ce problème sera complétement documenter en LaTex. Pour ce fair, nous devons prendre compte de plusieurs contraintes : ne pouvions pas utiliser de tableau intermédiaire et nous avions l'obligation d'utiliser uniquement des boucles de type while.

2 Formalisation du Problème

Notations clés :
$T[0 \dots k-1]$ — préfixe de la longueur k du tableau T
$T[N-k \dots N-1]$ — suffixe de la longueur k du tableau T

Prédicat :
$\text{pref\_equals\_suff}(T, N, k) \equiv (0 < k < N) \wedge \forall i \in [0 \dots k-1], T[i] = T[N-k+i]$

Fonction :
$\text{prefixe\_suffixe}(T, N) \equiv \max\{k \mid k \in [0 \dots N-1], \text{pref\_equals\_suff}(T, N, k)\}$

3 Définition et Analyse du Problème

3.1 Input/Output

— **Input** : Un tableau d'entiers T de taille N :
$T = (T[0], T[1] \dots T[N-1])$
$\wedge$
$N \geq 0$

— **Output** : Un entier k représentant la longueur maximale des sous-tableaux (préfixe et suffixe) du tableau T.
$k < N \wedge \text{pref\_equals\_suff}(T, N, k)$

Si de tels sous-tableaux n'existent pas, renvoyez 0.

3.2 Découpe en sous-problèmes

— **SP 1** : Énumération des longueurs possibles pour les préfixes et suffixes
Nous parcourons toutes les longueurs possibles k (pour les préfixes et suffixes) du tableau T dans l'ordre décroissant de $N-1$ jusqu'à 1. Pour chaque longueur k, nous vérifions la correspondance en utilisant la méthode de comparaison (SP 2). Dès que nous trouvons une longueur k pour laquelle le préfixe et le suffixe correspondent, nous la retournons. Si aucune longueur ne convient, nous retournons 0.

— **SP 2** : Vérification de l'égalité entre préfixe et suffixe
Pour une longueur k donnée, nous vérifions si le préfixe et le suffixe de T sont identiques. Cette comparaison s'effectue élément par élément.

4 Specifications

4.1 SP1 : Recherche du plus grand prefixe-suffixe :

— **Précondition** : T pointer vers un tableau de longueur $N \wedge N \geq 0$
— **Postcondition** : $T = T_0 \wedge N = N_0$
— **Retour** : prefixe_suffixe(T, N)

4.2 SP2 : Vérification que le préfixe et sufixe de longueur k sont égaux :

— **Précondition** : T pointer vers un tableau de longueur $N \wedge 0 < N \wedge < k < N$
— **Postcondition** : $T = T_0 \wedge N = N_0$
— **Retour** : pref_equals_suff(T, N, k)

5 Invariants

5.1 SP1 :

Invariant formel :

$T = T_0 \wedge N = N_0$
$\wedge$
$0 < N$
$\wedge$
$0 < k < N$
$\wedge$
$\neg pref\_equal\_suff(T, N, k)$

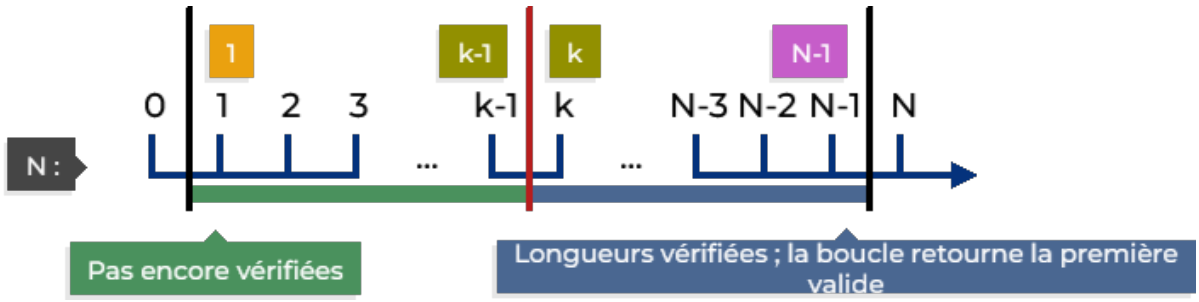

FIGURE 1 – Invariant graphique SP1

5.2 SP2 :

Invariant formel :

$T = T_0 \wedge N = N_0 \wedge k = k_0$
$\wedge$
$0 \leq i < k$
$\wedge$
$T[i] = T[N - k + i]$

6 Approche Constructive

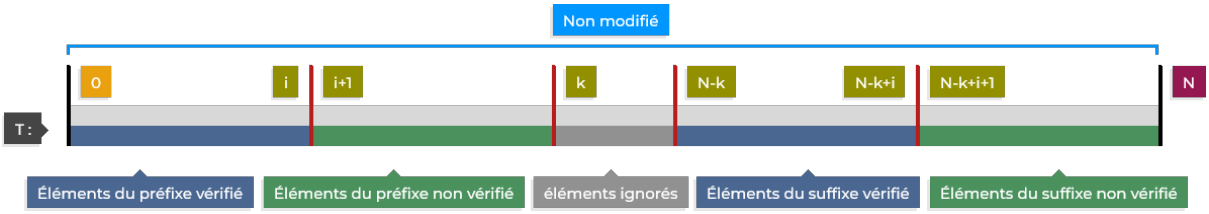

FIGURE 2 – Invariant graphique SP2

```c
int prefixe_suffixe(int *T, unsigned int N) {
    assert((T != NULL) && (0 < N));
    // 0 < N
    unsigned int k = N - 1;
    // k = N - 1
    while (0 < k && !pref_equal_suff(T, N, k)) {
        // 0 < k < N ∧ ¬pref_equal_suff(T, N, k)
        k--;
    }
    // k = 0 ∨ pref_equal_suff(T, N, k)
    return k;
    // T = T0 ∧ N = N0
}
```

Extrait de Code 1 – SP1

```c
static int pref_equal_suff(int *T, unsigned int N, unsigned int k) {
    assert((T != NULL) && (0 < N) && (0 < k && k < N));
    // 0 < k < N
    unsigned int i = 0;
    // i = 0
    while (i < k && T[i] == T[N - k + i]) {
        // i < k ∧ T[i] = T[N − k + i]
        i++;
    }
    // i = k ∨ T[i] ≠ T[N − k + i]
    return i == k;
    // T = T0 ∧ N = N0
}
```

Extrait de Code 2 – SP2

7 Code Complet

```c
#include <assert.h>
#include <stdlib.h>

#include "prefixe_suffixe.h"

/* ======= Private Function Prototypes ======= */

static int pref_equal_suff(int *T, unsigned int N, unsigned int k);

/* ======= Public Functions ======= */

/**
```

```
 * Sp 1
 * Checking all prefixes starting from the longest one until we find a match
 * or exhaust all possibilities.
 */
int prefixe_suffixe(int *T, unsigned int N) {
    assert((T != NULL) && (0 < N));
    // 0 < N
    unsigned int k = N - 1;
    // k = N - 1
    while (0 < k && !pref_equal_suff(T, N, k)) {
        // 0 < k < N ∧ ¬pref_equal_suff(T, N, k)
        k--;
    }
    // k = 0 ∨ pref_equal_suff(T, N, k)
    return k;
    // T = T0 ∧ N = N0
}

/* ======= Private Functions ======= */

/**
 * Sp 2
 * Comparing the prefix and suffix of the given length, element by element.
 */
static int pref_equal_suff(int *T, unsigned int N, unsigned int k) {
    assert((T != NULL) && (0 < N) && (0 < k && k < N));
    // 0 < k < N
    unsigned int i = 0;
    // i = 0
    while (i < k && T[i] == T[N - k + i]) {
        // i < k ∧ T[i] = T[N − k + i]
        i++;
    }
    // i = k ∨ T[i] ≠ T[N − k + i]
    return i == k;
    // T = T0 ∧ N = N0
}
```

Extrait de Code 3 – Implémentation de prefixe_suffixe

8 Complexité

Complexité de `pref_equal_suff` :

— Complexité exacte :
$T_1(k) = 1 + (k+1) + k + k + 1 = 3k + 3$

— Asymptotique :
$T_1(k) \in \mathcal{O}(k)$

Complexité de `prefixe_suffixe` :

— Complexité exacte :

$$T_2(N) = 1 + \sum_{k=N-1}^{1} [3 + T_1(k)] + 1 = 2 + 3 \sum_{k=N-1}^{1} [3k + 6] = 2 + \frac{3(N-1)N}{2} + 6(N-1) = \frac{3N^2 - 3N}{2} + 6N - 4$$

— Asymptotique :
$T_2(N) \in \mathcal{O}(N^2)$

9 Conclusion

Pour terminer ce rapport nous pouvons ajouter que nous avons réussi à créer un algorithme en C cappable de trouver le plus long préfixe qui est aussi un suffixe d'une liste donnée. Et tout cela en respectant les contraintes données tel que l'utilisation exclusives des boucles while et l'abscence de l'utilisation de list intermédiaires.